Chapter 8. Semantic Search and Retrieval-Augmented Generation

Search was one of the first language model applications to see broad industry adoption. Months after the release of the seminal "BERT: Pre-training of deep bidirectional transformers for language understanding" (2018) paper, Google announced it was using it to power Google Search and that it represented "one of the biggest leaps forward in the history of Search." Not to be outdone, Microsoft Bing also stated that "Starting from April of this year, we used large transformer models to deliver the largest quality improvements to our Bing customers in the past year."

This is a clear testament to the power and usefulness of these models. Their addition instantly and dramatically improves some of the most mature, well-maintained systems that billions of people around the planet rely on. The ability they add is called *semantic search*, which enables searching by meaning, and not simply keyword matching.

On a separate track, the fast adoption of text generation models led many users to ask the models questions and expect factual answers. And while the models were able to answer fluently and confidently, their answers were not always correct or up-to-date. This problem grew to be known as model "hallucinations," and one of the leading ways to reduce it is to build systems that can retrieve relevant information and provide it to the LLM to aid it in generating more factual answers. This method, called RAG, is one of the most popular applications of LLMs.

Overview of Semantic Search and RAG

There's a lot of research on how to best use language models for search. Three broad categories of these models are dense retrieval, reranking, and RAG. Here is an overview of these three categories that the rest of the chapter will then explain in more detail:

Dense retrieval

Dense retrieval systems rely on the concept of embeddings, the same concept we've encountered in the previous chapters, and turn the search problem into retrieving the nearest neighbors of the search query (after both the query and the documents are converted into embeddings). Figure 8-1 shows how dense retrieval takes a search query, consults its archive of texts, and outputs a set of relevant results.

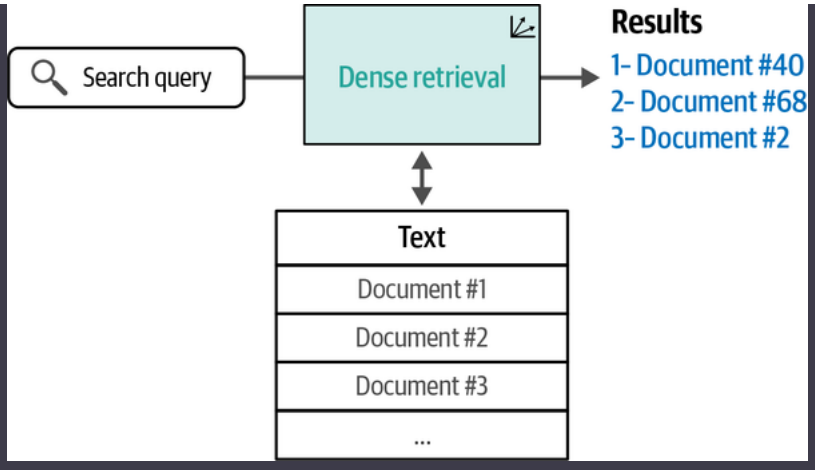

Figure 8-1. Dense retrieval is one of the key types of semantic search, relying on the similarity of text embeddings to retrieve relevant results.

Reranking

Search systems are often pipelines of multiple steps. A reranking language model is one of these steps and is tasked with scoring the relevance of a subset of results against the query; the order of results is then changed based on these scores. Figure 8-2 shows how rerankers are different from dense retrieval in that they take an additional input: a set of search results from a previous step in the search pipeline.

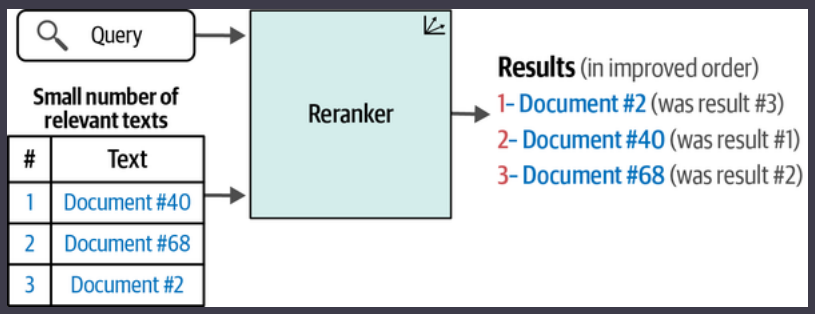

Figure 8-2. Rerankers, the second key type of semantic search, take a search query and a collection of results, and reorder them by relevance, often resulting in vastly improved results.

RAG

The growing LLM capability of text generation led to a new type of search systems that include a model that generates an answer in response to a query. Figure 8-3 shows an example of such a generative search system.

Generative search is a subset of a broader type of category of systems better called RAG systems. These are text generation systems that incorporate search capabilities to reduce hallucinations, increase factuality, and/or ground the generation model on a specific dataset.

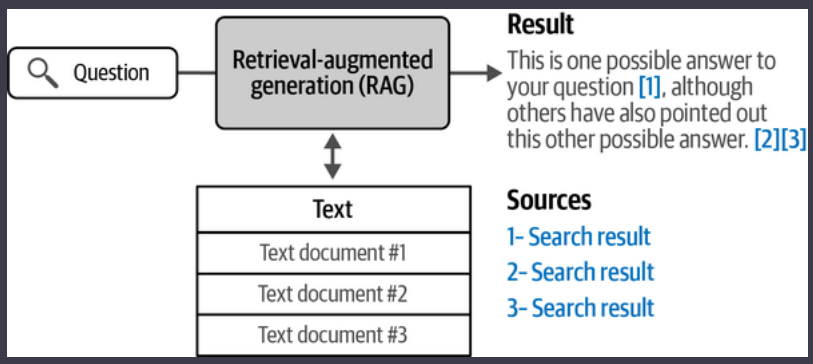

Figure 8-3. A RAG system formulates an answer to a question and (preferably) cites its information sources.

The rest of the chapter covers these three types of systems in more detail. While these are the major categories, they are not the only LLM applications in the domain of search.

Semantic Search with Language Models

Let's now dive into more detail on the major categories of systems that can upgrade the search capabilities of our language models. We'll start with dense retrieval and then move on through reranking and RAG.

Dense Retrieval

Recall that embeddings turn text into numeric representations. Those can be thought of as points in space, as we can see in Figure 8-4. Points that are close together mean that the text they represent is similar. So in this example, text 1 and text 2 are more similar to each other (because they are near each other) than text 3 (because it's farther away).

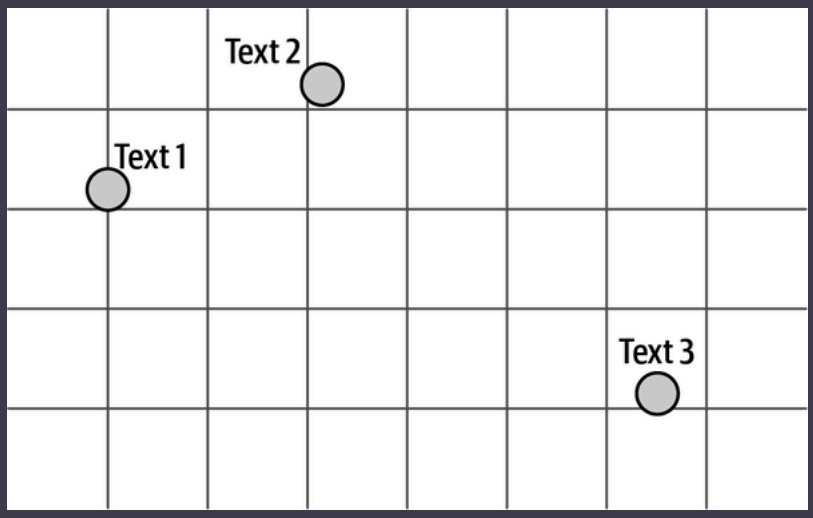

Figure 8-4. The intuition of embeddings: each text is a point and texts with similar meaning are close to each other.

This is the property that is used to build search systems. In this scenario, when a user enters a search query, we embed the query, thus projecting it into the same space as our text archive. Then we simply find the nearest documents to the query in that space, and those would be the search results (Figure 8-5).

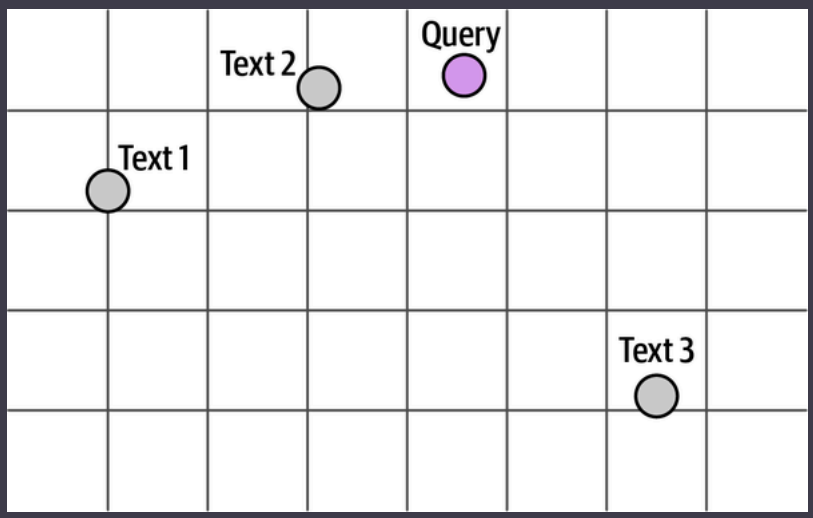

Figure 8-5. Dense retrieval relies on the property that search queries will be close to their relevant results.

Judging by the distances in Figure 8-5, "text 2" is the best result for this query, followed by "text 1." Two questions could arise here, however:

- Should text 3 even be returned as a result? That's a decision for you, the system designer. It's sometimes desirable to have a max threshold of similarity score to filter out irrelevant results (in case the corpus has no relevant results for the query).
- Are a query and its best result semantically similar? Not always. This is why language models need to be trained on question-answer pairs to become better at retrieval. This process is explained in more detail in Chapter 10.

Figure 8-6 shows how we chunk a document before proceeding to embed each chunk. Those embedding vectors are then stored in the vector database and are ready for retrieval.

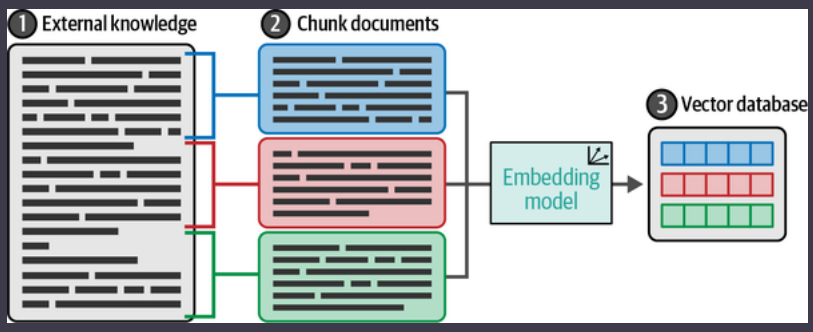

Figure 8-6. Convert some external knowledge base to a vector database. We can then query this vector database for information about the knowledge base.

Dense retrieval example

Let's take a look at a dense retrieval example by using Cohere to search the Wikipedia page for the film *Interstellar*. In this example, we will do the following:

1. Get the text we want to make searchable and apply some light processing to chunk it into sentences.
2. Embed the sentences.
3. Build the search index.
4. Search and see the results.

Get your Cohere API key by signing up at *https://oreil.ly/GxrQ1*. Paste it in the following code. You will not have to pay anything to run through this example.

Let's import the libraries we'll need:

```
import cohere
import numpy as np
import pandas as pd
from tqdm import tqdm

# Paste your API key here. Remember to not share publicly
api_key = ''

# Create and retrieve a Cohere API key from os.cohere.ai
co = cohere.Client(api_key)
```

Getting the text archive and chunking it

Let's use the first section of the Wikipedia article on the film *Interstellar*. We'll get the text, then break it into sentences:

```
text = """
Interstellar is a 2014 epic science fiction film co-written, directed, and produced
It stars Matthew McConaughey, Anne Hathaway, Jessica Chastain, Bill Irwin, Ellen Bu
Set in a dystopian future where humanity is struggling to survive, the film follows

Brothers Christopher and Jonathan Nolan wrote the screenplay, which had its origins
Caltech theoretical physicist and 2017 Nobel laureate in Physics[4] Kip Thorne was
Cinematographer Hoyte van Hoytema shot it on 35 mm movie film in the Panavision ana
Principal photography began in late 2013 and took place in Alberta, Iceland, and Lo
Interstellar uses extensive practical and miniature effects and the company Double

Interstellar premiered on October 26, 2014, in Los Angeles.
In the United States, it was first released on film stock, expanding to venues usi
The film had a worldwide gross over $677 million (and $773 million with subsequent
It received acclaim for its performances, direction, screenplay, musical score, vis
It has also received praise from many astronomers for its scientific accuracy and p
Interstellar was nominated for five awards at the 87th Academy Awards, winning Best

# Split into a list of sentences
texts = text.split('.')

# Clean up to remove empty spaces and new lines
texts = [t.strip(' \n') for t in texts]
```

Embedding the text chunks

Let's now embed the texts. We'll send them to the Cohere API, and get back a vector for each text:

```
# Get the embeddings
response = co.embed(
  texts=texts,
  input_type="search_document",
).embeddings

embeds = np.array(response)
print(embeds.shape)
```

This outputs `(15, 4096)`, which indicates that we have 15 vectors, each one of size 4,096.

Building the search index

Before we can search, we need to build a search index. An index stores the embeddings and is optimized to quickly retrieve the nearest neighbors even if we have a very large number of

points:

```
import faiss
dim = embeds.shape[1]
index = faiss.IndexFlatL2(dim)
print(index.is_trained)
index.add(np.float32(embeds))
```

Search the index

We can now search the dataset using any query we want. We simply embed the query and present its embedding to the index, which will retrieve the most similar sentence from the Wikipedia article.

Let's define our search function:

```
def search(query, number_of_results=3):

  # 1. Get the query's embedding
  query_embed = co.embed(texts=[query],
                input_type="search_query",).embeddings[0]

  # 2. Retrieve the nearest neighbors
  distances , similar_item_ids = index.search(np.float32([query_embed]), number_of

  # 3. Format the results
  texts_np = np.array(texts) # Convert texts list to numpy for easier indexing
  results = pd.DataFrame(data={'texts': texts_np[similar_item_ids[0]],
                               'distance': distances[0]})

  # 4. Print and return the results
  print(f"Query:'{query}'\nNearest neighbors:")
  return results
```

We are now ready to write a query and search the texts!

```
query = "how precise was the science"
results = search(query)
results
```

This produces the following output:

```
Query: 'how precise was the science'
Nearest neighbors:
```

	texts	distance
0	It has also received praise from many astronomers for its scientific accuracy and portrayal of theoretical astrophysics	10757.379883
1	Caltech theoretical physicist and 2017 Nobel laureate in Physics[4] Kip Thorne was an executive producer, acted as a scientific consultant, and wrote a tie-in book, The Science of Interstellar	11566.131836
2	Interstellar uses extensive practical and miniature effects and the company Double Negative created additional digital effects	11922.833008

The first result has the least distance, and so is the most similar to the query. Looking at it, it answers the question perfectly. Notice that this wouldn't have been possible if we were only doing keyword search because the top result did not include the same keywords in the query.

We can actually verify that by defining a keyword search function to compare the two. We'll use the BM25 algorithm, which is one of the leading lexical search methods. See this notebook for the source of these code snippets:

```
from rank_bm25 import BM25Okapi
from sklearn.feature_extraction import _stop_words
import string

def bm25_tokenizer(text):
    tokenized_doc = []
    for token in text.lower().split():
        token = token.strip(string.punctuation)

        if len(token) > 0 and token not in _stop_words.ENGLISH_STOP_WORDS:
            tokenized_doc.append(token)
    return tokenized_doc


tokenized_corpus = []
for passage in tqdm(texts):
    tokenized_corpus.append(bm25_tokenizer(passage))

bm25 = BM25Okapi(tokenized_corpus)

def keyword_search(query, top_k=3, num_candidates=15):
    print("Input question:", query)

    ##### BM25 search (lexical search) #####
    bm25_scores = bm25.get_scores(bm25_tokenizer(query))
    top_n = np.argpartition(bm25_scores, -num_candidates)[-num_candidates:]
    bm25_hits = [{'corpus_id': idx, 'score': bm25_scores[idx]} for idx in top_n]
    bm25_hits = sorted(bm25_hits, key=lambda x: x['score'], reverse=True)

    print(f"Top-3 lexical search (BM25) hits")
```

```
    for hit in bm25_hits[0:top_k]:
        print("\t{:.3f}\t{}".format(hit['score'], texts[hit['corpus_id']].replace('
```

Now when we search for the same query, we get a different set of results from the dense retrieval search:

```
keyword_search(query = "how precise was the science")
```

Results:

```
Input question: how precise was the science
Top-3 lexical search (BM25) hits
      1.789 Interstellar is a 2014 epic science fiction film co-written,
      1.373 Caltech theoretical physicist and 2017 Nobel laureate in Phys
      0.000 It stars Matthew McConaughey, Anne Hathaway, Jessica Chastain
```

Note that the first result does not really answer the question despite it sharing the word "science" with the query. In the next section, we'll see how adding a reranker can improve this search system. But before that, let's complete our overview of dense retrieval by looking at its caveats and go over some methods of breaking down texts into chunks.

Caveats of dense retrieval

It's useful to be aware of some of the drawbacks of dense retrieval and how to address them. What happens, for example, if the texts don't contain the answer? We still get results and their distances. For example:

```
Query:'What is the mass of the moon?'
Nearest neighbors:
```

	texts	distance
0	The film had a worldwide gross over $677 million (and $773 million with subsequent re-releases), making it the tenth-highest grossing film of 2014	1.298275
1	It has also received praise from many astronomers for its scientific accuracy and portrayal of theoretical astrophysics	1.324389
2	Cinematographer Hoyte van Hoytema shot it on 35 mm movie film in the Panavision anamorphic format and IMAX 70 mm	1.328375

In cases like this, one possible heuristic is to set a threshold level—a maximum distance for relevance, for example. A lot of search systems present the user with the best info they can get and leave it up to the user to decide if it's relevant or not. Tracking the information of whether

the user clicked on a result (and were satisfied by it) can improve future versions of the search system.

Another caveat of dense retrieval is when a user wants to find an exact match for a specific phrase. That's a case that's perfect for keyword matching. That's one reason why hybrid search, which includes both semantic search and keyword search, is advised instead of relying solely on dense retrieval.

Dense retrieval systems also find it challenging to work properly in domains other than the ones that they were trained on. So, for example, if you train a retrieval model on internet and Wikipedia data, and then deploy it on legal texts (without having enough legal data as part of the training set), the model will not work as well in that legal domain.

The final thing we'd like to point out is that this is a case where each sentence contained a piece of information, and we showed queries that specifically ask for that information. What about questions whose answers span multiple sentences? This highlights one of the important design parameters of dense retrieval systems: what is the best way to chunk long texts? And why do we need to chunk them in the first place?

Chunking long texts

One limitation of Transformer language models is that they are limited in context sizes, meaning we cannot feed them very long texts that go above the number of words or tokens that the model supports. So how do we embed long texts?

There are several possible ways, and two possible approaches shown in Figure 8-7 include indexing one vector per document and indexing multiple vectors per document.

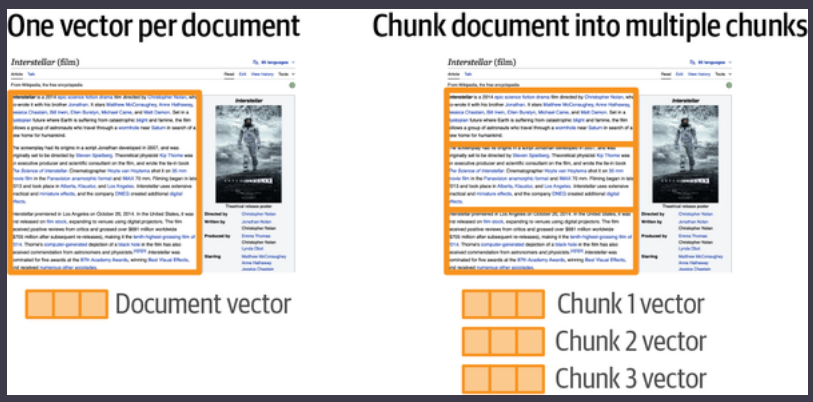

Figure 8-7. It's possible to create one vector representing an entire document, but it's better for longer documents to be split into smaller chunks that get their own embeddings.

One vector per document

In this approach, we use a single vector to represent the whole document. The possibilities here include:

- Embedding only a representative part of the document and ignoring the rest of the text. This may mean embedding only the title, or only the beginning of the document. This is useful to get quickly started with building a demo but it leaves a lot of information unindexed and therefore unsearchable. As an approach, it may work better for documents where the beginning captures the main points of a document (think: Wikipedia article). But it's really not the

best approach for a real system because a lot of information would be left out of the index and would be unsearchable.

- Embedding the document in chunks, embedding those chunks, and then aggregating those chunks into a single vector. The usual method of aggregation here is to average those vectors. A downside of this approach is that it results in a highly compressed vector that loses a lot of the information in the document.

This approach can satisfy some information needs, but not others. A lot of the time, a search is for a specific piece of information contained in an article, which is better captured if the concept had its own vector.

Multiple vectors per document

In this approach, we chunk the document into smaller pieces, and embed those chunks. Our search index then becomes that of chunk embeddings, not entire document embeddings. Figure 8-8 shows a number of possible text chunking approaches.

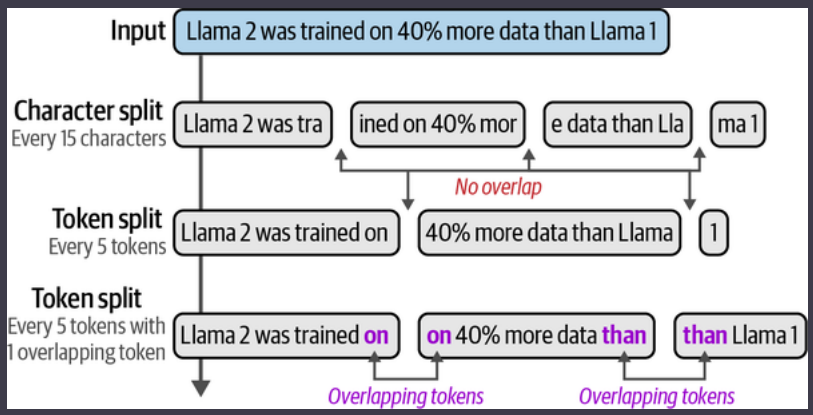

Figure 8-8. Several chunking methods and their effects on the input text. Overlapping chunks can be important to prevent the absence of context.

The chunking approach is better because it has full coverage of the text and because the vectors tend to capture individual concepts inside the text. This leads to a more expressive search index. Figure 8-9 shows a number of possible approaches.

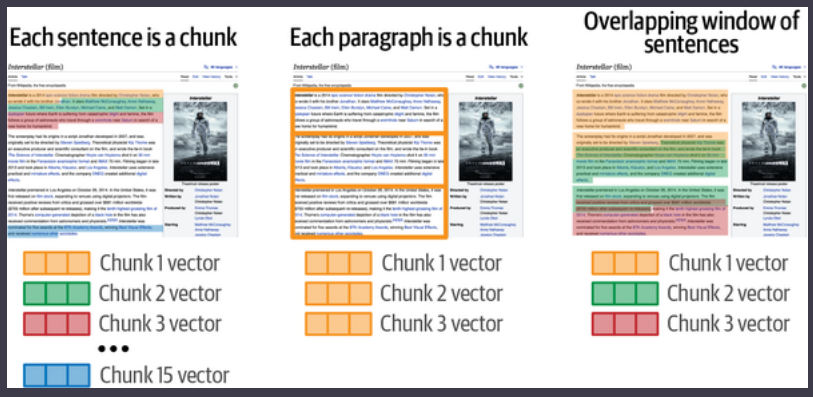

Figure 8-9. A number of possible options for chunking a document for embedding.

The best way of chunking a long text will depend on the types of texts and queries your system anticipates. Approaches include:

- Each sentence is a chunk. The issue here is this could be too granular and the vectors don't capture enough of the context.

- Each paragraph is a chunk. This is great if the text is made up of short paragraphs. Otherwise, it may be that every 3–8 sentences is a chunk.
- Some chunks derive a lot of their meaning from the text around them. So we can incorporate some context via:
  - Adding the title of the document to the chunk.
  - Adding some of the text before and after them to the chunk. This way, the chunks can overlap so they include some surrounding text that also appears in adjacent chunks. This is what we can see in Figure 8-10.

Expect more chunking strategies to arise as the field develops—some of which may even use LLMs to dynamically split a text into meaningful chunks.

Figure 8-10. Chunking the text into overlapping segments is one strategy to retain more of the context around different segments.

Nearest neighbor search versus vector databases

Once the query is embedded, we need to find the nearest vectors to it from our text archive as we can see in Figure 8-11. The most straightforward way to find the nearest neighbors is to calculate the distances between the query and the archive. That can easily be done with NumPy and is a reasonable approach if you have thousands or tens of thousands of vectors in your archive.

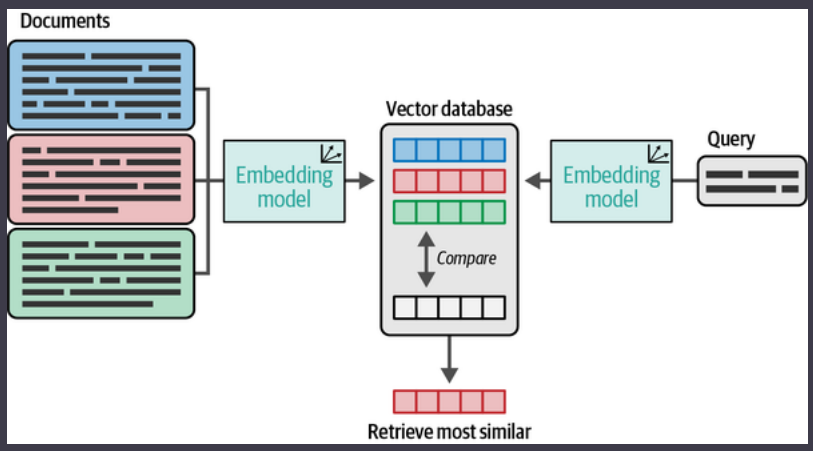

Figure 8-11. As we saw in Chapter 3, we can compare embeddings to quickly find the most similar documents to a query.

As you scale beyond to the millions of vectors, an optimized approach for retrieval is to rely on approximate nearest neighbor search libraries like Annoy or FAISS. These allow you to retrieve results from massive indexes in milliseconds and some of them can improve their performance by utilizing GPUs and scaling to clusters of machines to serve very large indices.

Another class of vector retrieval systems are vector databases like Weaviate or Pinecone. A vector database allows you to add or delete vectors without having to rebuild the index. They also provide ways to filter your search or customize it in ways beyond merely vector distances.

Fine-tuning embedding models for dense retrieval

Just as we discussed in Chapter 4 on text classification, we can improve the performance of an LLM on a task using fine-tuning. As in that case, retrieval needs to optimize text embeddings and not simply token embeddings. The process for this fine-tuning is to get training data composed of queries and relevant results.

Let's look at one example from our dataset, the sentence "Interstellar premiered on October 26, 2014, in Los Angeles." Two possible queries where this is a relevant result are:

- Relevant query 1: "Interstellar release date"
- Relevant query 2: "When did Interstellar premier"

The fine-tuning process aims to make the embeddings of these queries close to the embedding of the resulting sentence. It also needs to see negative examples of queries that are not relevant to the sentence, for example:

- Irrelevant query: "Interstellar cast"

With these examples, we now have three pairs—two positive pairs and one negative pair. Let's assume, as we can see in Figure 8-12, that before fine-tuning, all three queries have the same distance from the result document. That's not far-fetched because they all talk about *Interstellar*.

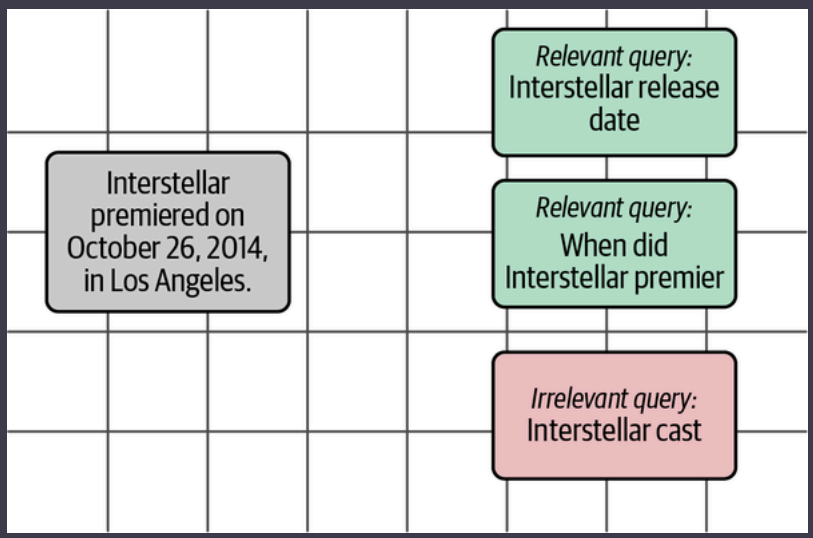

Figure 8-12. Before fine-tuning, the embeddings of both relevant and irrelevant queries may be close to a particular document.

The fine-tuning step works to make the relevant queries closer to the document and at the same time make irrelevant queries farther from the document. We can see this effect in Figure 8-13.

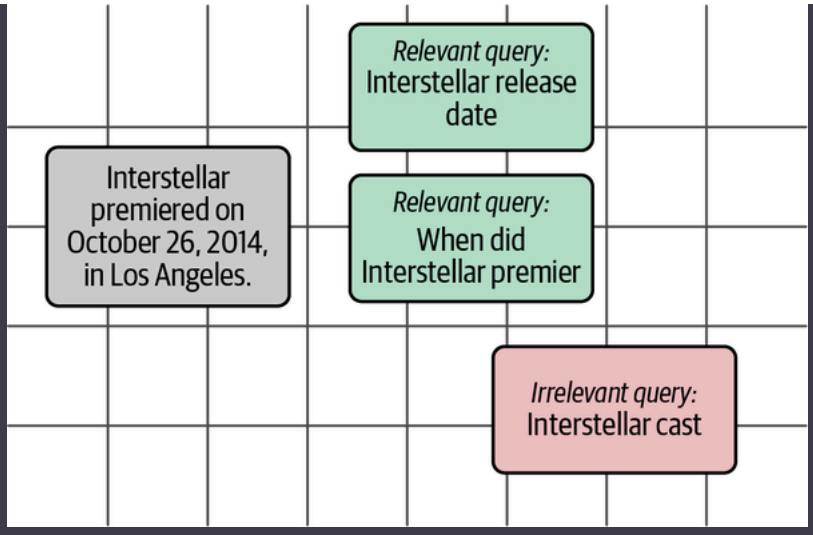

Figure 8-13. After the fine-tuning process, the text embedding model becomes better at this search task by incorporating how we define relevance on our dataset using the examples we provided of relevant and irrelevant documents.

Reranking

A lot of organizations have already built search systems. For those organizations, an easier way to incorporate language models is as a final step inside their search pipeline. This step is tasked with changing the order of the search results based on relevance to the search query. This one step can vastly improve search results and it's in fact what Microsoft Bing added to achieve the improvements to search results using BERT-like models. Figure 8-14 shows the structure of a rerank search system serving as the second stage in a two-stage search system.

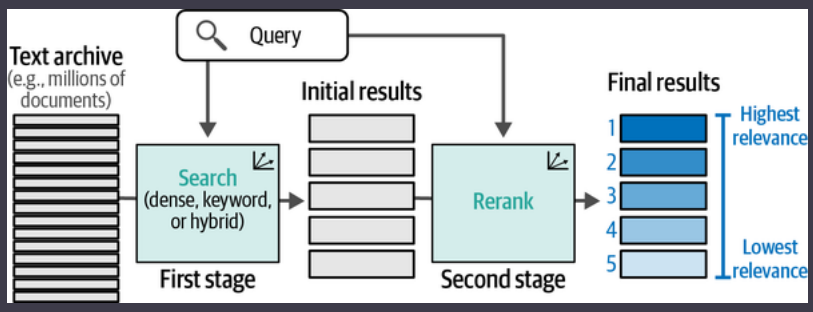

Figure 8-14. LLM rerankers operate as part of a search pipeline with the goal of reordering a number of shortlisted search results by relevance.

Reranking example

A reranker takes in the search query and a number of search results, and returns the optimal ordering of these documents so the most relevant ones to the query are higher in ranking. Cohere's Rerank endpoint is a simple way to start using a first reranker. We simply pass it the query and texts and get the results back. We don't need to train or tune it:

```
query = "how precise was the science"
results = co.rerank(query=query, documents=texts, top_n=3, return_documents=True)
results.results
```

We can print these results:

```
for idx, result in enumerate(results.results):
```

```
    print(idx, result.relevance_score , result.document.text)
```

Output:

```
0 0.1698185 It has also received praise from many astronomers for its sci
1 0.07004896 The film had a worldwide gross over $677 million (and $773 m
2 0.0043994132 Caltech theoretical physicist and 2017 Nobel laureate in P
```

This shows the reranker is much more confident about the first result, assigning it a relevance score of 0.16, while the other results are scored much lower in relevance.

In this basic example, we passed our reranker all 15 of our documents. More often, however, our index would have thousands or millions of entries, and we need to shortlist, say one hundred or one thousand results and then present those to the reranker. This shortlisting step is called the *first stage* of the search pipeline.

The first-stage retriever can be keyword search, dense retrieval, or better yet—hybrid search that uses both of them. We can revisit our previous example to see how adding a reranker after a keyword search system improves its performance.

Let's tweak our keyword search function so it retrieves a list of the top 10 results using keyword search, then use rerank to choose the top 3 results from those 10:

```
def keyword_and_reranking_search(query, top_k=3, num_candidates=10):
    print("Input question:", query)

    ##### BM25 search (lexical search) #####
    bm25_scores = bm25.get_scores(bm25_tokenizer(query))
    top_n = np.argpartition(bm25_scores, -num_candidates)[-num_candidates:]
    bm25_hits = [{'corpus_id': idx, 'score': bm25_scores[idx]} for idx in top_n]
    bm25_hits = sorted(bm25_hits, key=lambda x: x['score'], reverse=True)

    print(f"Top-3 lexical search (BM25) hits")
    for hit in bm25_hits[0:top_k]:
        print("\t{:.3f}\t{}".format(hit['score'], texts[hit['corpus_id']].replace('


    #Add re-ranking
    docs = [texts[hit['corpus_id']] for hit in bm25_hits]

    print(f"\nTop-3 hits by rank-API ({len(bm25_hits)} BM25 hits re-ranked)")
    results = co.rerank(query=query, documents=docs, top_n=top_k, return_documents=
    # print(results.results)
    for hit in results.results:
        # print(hit)
        print("\t{:.3f}\t{}".format(hit.relevance_score, hit.document.text.replace
```

Now we can send our query and check the results of keyword search and then the result of keyword search shortlisting its top 10 results, then pass them on to the reranker:

```
keyword_and_reranking_search(query = "how precise was the science")
```

Results:

```
Input question: how precise was the science
Top-3 lexical search (BM25) hits
1.789 Interstellar is a 2014 epic science fiction film co-written, direct
1.373 Caltech theoretical physicist and 2017 Nobel laureate in Physics[4]
0.000 Interstellar uses extensive practical and miniature effects and the

Top-3 hits by rank-API (10 BM25 hits re-ranked)
0.004 Caltech theoretical physicist and 2017 Nobel laureate in Physics[4]
0.004 Set in a dystopian future where humanity is struggling to survive,
0.003 Brothers Christopher and Jonathan Nolan wrote the screenplay, which
```

We see that keyword search assigns scores to only two results that share some of the keywords. In the second set of results, the reranker elevates the second result appropriately as the most relevant result for the query. This is a toy example that gives us a glimpse of the effect, but in practice, such a pipeline significantly improves search quality. On a multilingual benchmark like MIRACL, a reranker can boost performance from 36.5 to 62.8, measured as nDCG@10 (more on evaluation later in this chapter).

Open source retrieval and reranking with sentence transformers

If you want to locally set up retrieval and reranking on your own machine, then you can use the Sentence Transformers library. Refer to the documentation at *https://oreil.ly/jJOhV* for set-up. Check the "Retrieve & Re-Rank" section for instructions and code examples for how to conduct these steps in the library.

How reranking models work

One popular way of building LLM search rerankers is to present the query and each result to an LLM working as a *cross-encoder*. This means that a query and possible result are presented to the model at the same time allowing the model to view both these texts before it assigns a relevance score, as we can see in Figure 8-15. All of the documents are processed simultaneously as a batch yet each document is evaluated against the query independently. The scores then determine the new order of the results. This method is described in more detail in a paper titled "Multi-stage document ranking with BERT" and is sometimes referred to as mono-BERT.

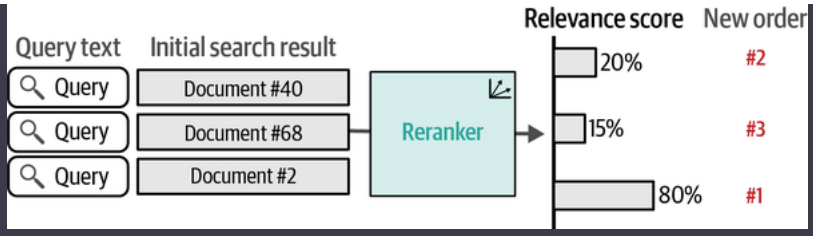

Figure 8-15. A reranker assigns a relevance score to each document by looking at the document and the query at the same time.

This formulation of search as relevance scoring basically boils down to being a classification problem. Given those inputs, the model outputs a score from 0–1 where 0 is irrelevant and 1 is highly relevant. This should be familiar from our classification discussions in Chapter 4.

To learn more about the development of using LLMs for search, "Pretrained transformers for text tanking: BERT and beyond" is a highly recommended look at the developments of these models until about 2021.

Retrieval Evaluation Metrics

Semantic search is evaluated using metrics from the Information Retrieval (IR) field. Let's discuss one of these popular metrics: mean average precision (MAP).

Evaluating search systems needs three major components: a text archive, a set of queries, and relevance judgments indicating which documents are relevant for each query. We see these components in Figure 8-16.

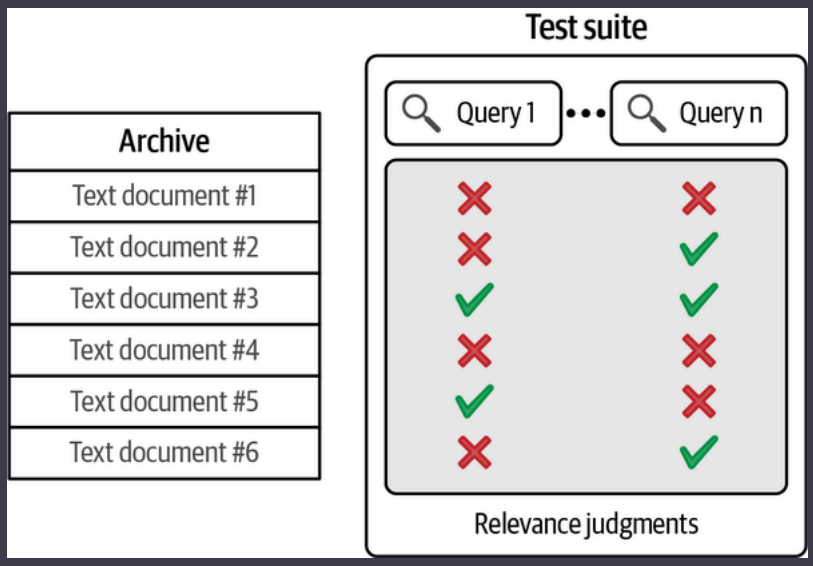

Figure 8-16. To evaluate search systems, we need a test suite including queries and relevance judgments indicating which documents in our archive are relevant for each query.

Using this test suite, we can proceed to explore evaluating search systems. Let's start with a simple example. Let's assume we pass query 1 to two different search systems. And get two sets of results. Say we limit the number of results to three, as we can see in Figure 8-17.

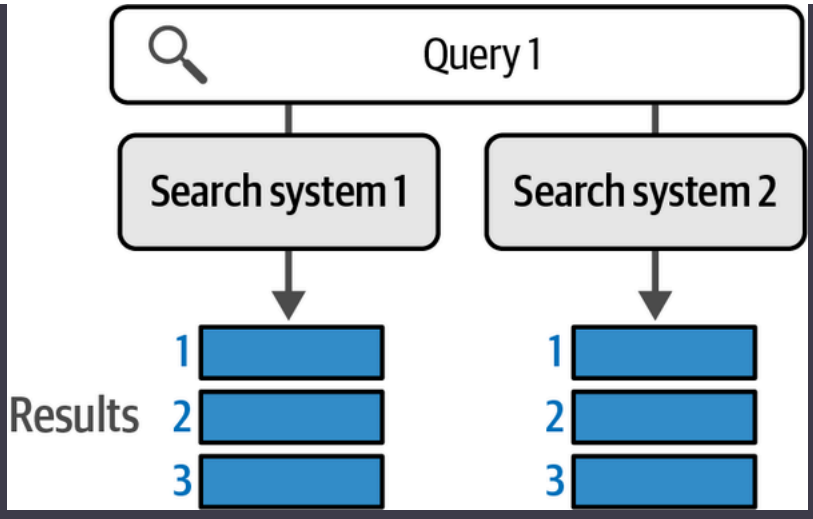

Figure 8-17. To compare two search systems, we pass the same query from our test suite to both systems and look at their top results.

To tell which is a better system, we turn to the relevance judgments that we have for the query. Figure 8-18 shows which of the returned results are relevant.

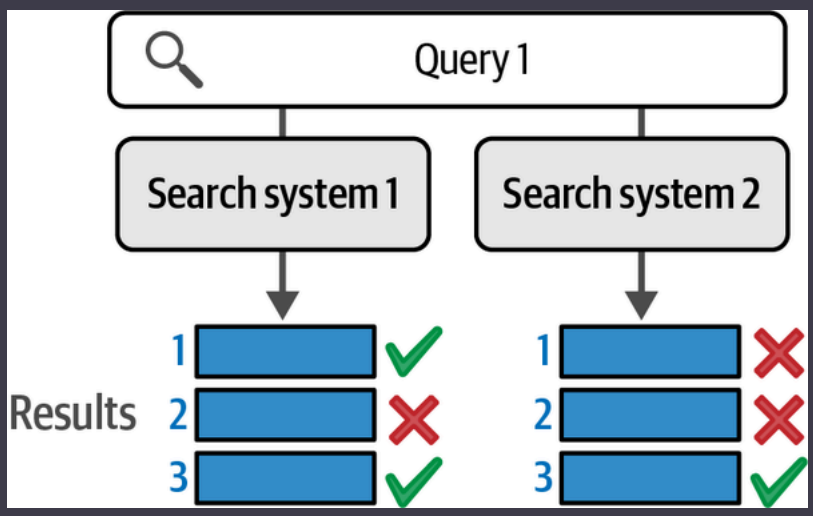

Figure 8-18. Looking at the relevance judgments from our test suite, we can see that system 1 did a better job than system 2.

This shows us a clear case where system 1 is better than system 2. Intuitively, we may just count how many relevant results each system retrieved. System 1 got two out of three correct, and system 2 got only one out of three correct. But what about a case like Figure 8-19 where both systems only get one relevant result out of three, but they're in different positions?

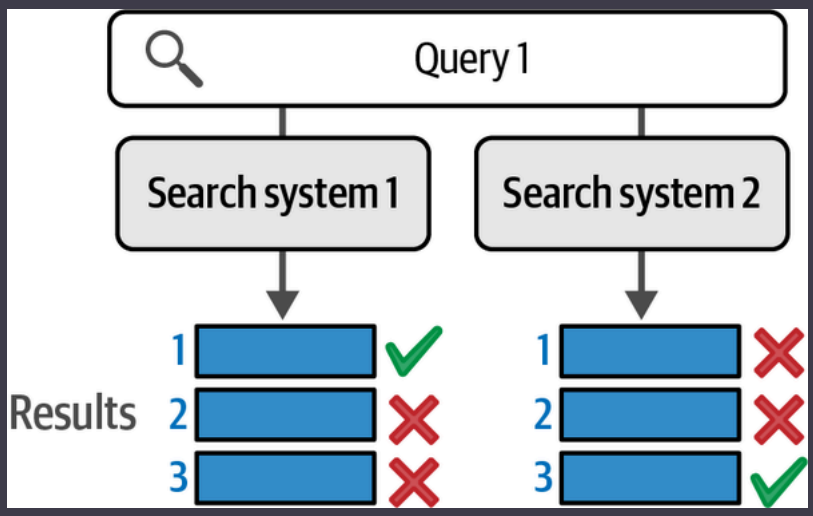

Figure 8-19. We need a scoring system that rewards system 1 for assigning a high position to a relevant result—even though both systems retrieved only one relevant result in their top three results.

In this case, we can intuit that system 1 did a better job than system 2 because the result in the first position (the most important position) is correct. But how can we assign a number or score to how much better that result is? Mean average precision is a measure that is able to quantify this distinction.

One common way to assign numeric scores in this scenario is average precision, which evaluates system 1's result for the query to be 1 and system 2's to be 0.3. So let's see how average precision is calculated to evaluate one set of results, and then how it's aggregated to evaluate a system across all the queries in the test suite.

Scoring a single query with average precision

To score a search system on this query, we can focus on scoring the relevant documents. Let's start by looking at a query that only has one relevant document in the test suite.

The first one is easy: the search system placed the relevant result (the only available one for this query) at the top. This gets the system the perfect score of 1. Figure 8-20 shows this calculation: looking at the first position, we have a relevant result leading to a precision at position 1 of 1.0 (calculated as the number of relevant results at position 1, divided by the position we're currently looking at).

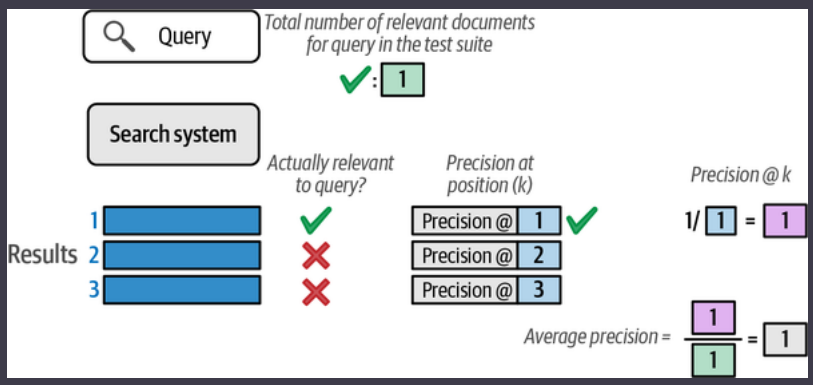

Figure 8-20. To calculate mean average precision, we start by calculating precision at each position, starting with position 1.

Since we're only scoring relevant documents we can ignore the scores of nonrelevant documents and stop our calculation here. What if the system actually placed the only relevant result at the third position, however? How would that affect the score? Figure 8-21 shows how that results in a penalty.

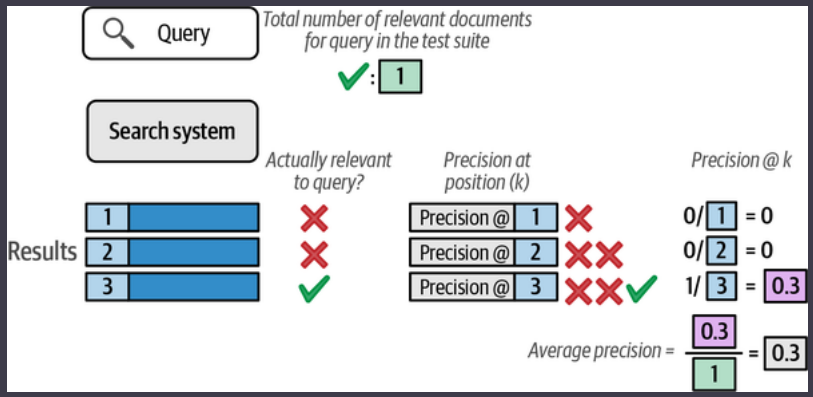

Figure 8-21. If the system places nonrelevant documents ahead of a relevant document, its precision score is penalized.

Let's now look at a query with more than one relevant document. Figure 8-22 shows that calculation and how averaging now comes into the picture.

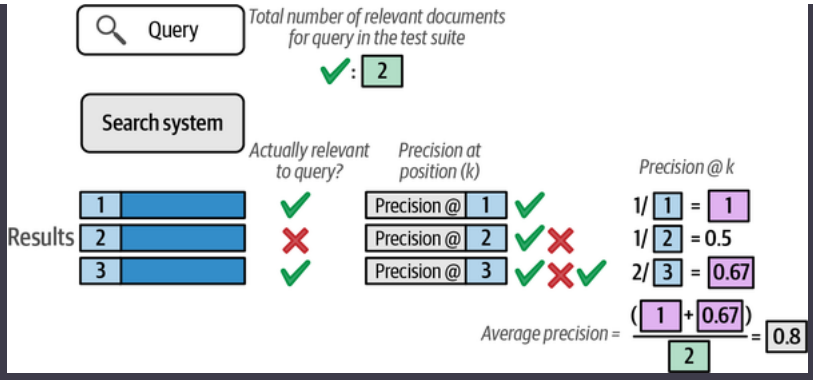

Figure 8-22. Average precision of a document with multiple relevant documents considers the precision at k results of all the relevant documents.

Scoring across multiple queries with mean average precision

Now that we're familiar with precision at k and average precision, we can extend this knowledge to a metric that can score a search system against all the queries in our test suite. That metric is called mean average precision. Figure 8-23 shows how to calculate this metric by taking the mean of the average precisions of each query.

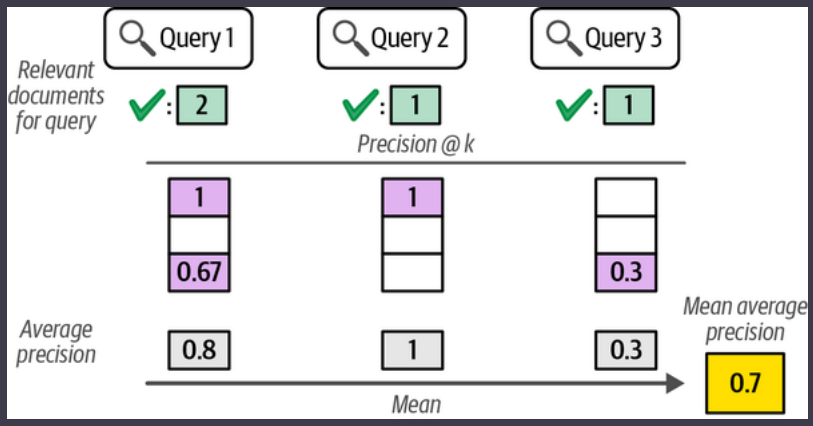

Figure 8-23. The mean average precision takes into consideration the average precision score of a system for every query in the test suite. By averaging them, it produces a single metric that we can use to compare a search system against another.

You may be wondering why the same operation is called "mean" and "average." It's likely an aesthetic choice because MAP sounds better than average average precision.

Now we have a single metric that we can use to compare different systems. If you want to learn more about evaluation metrics, see the "Evaluation in Information Retrieval" chapter of *Introduction to Information Retrieval* (Cambridge University Press) by Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze.

In addition to mean average precision, another metric commonly used for search systems is normalized discounted cumulative gain (nDCG), which is more nuanced in that the relevance of documents is not binary (relevant versus not relevant) and one document can be labeled as more relevant than another in the test suite and scoring mechanism.

Retrieval-Augmented Generation (RAG)

The mass adoption of LLMs quickly led to people asking them questions and expecting factual answers. While the models can answer some questions correctly, they also confidently answer lots of questions incorrectly. The leading method the industry turned to remedy this behavior

is RAG, described in the paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" (2020)[1] and illustrated in Figure 8-24.

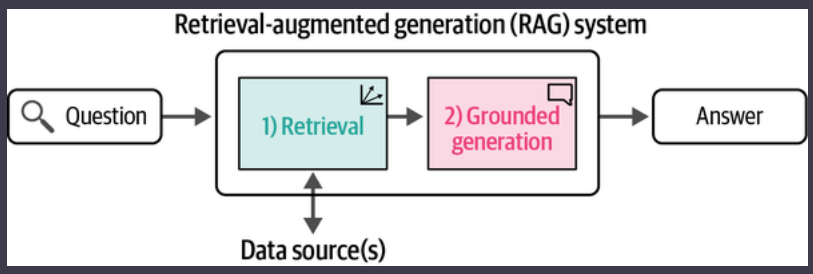

Figure 8-24. A basic RAG pipeline is made up of a search step followed by a grounded generation step where the LLM is prompted with the question and the information retrieved from the search step.

RAG systems incorporate search capabilities in addition to generation capabilities. They can be seen as an improvement to generation systems because they reduce their hallucinations and improve their factuality. They also enable use cases of "chat with my data" that consumers and companies can use to ground an LLM on internal company data, or a specific data source of interest (e.g., chatting with a book).

This also extends to search systems. More search engines are incorporating an LLM to summarize results or answer questions submitted to the search engine. Examples include Perplexity, Microsoft Bing AI, and Google Gemini.

From Search to RAG

Let's now turn our search system into a RAG system. We do that by adding an LLM to the end of the search pipeline. We present the question and the top retrieved documents to the LLM, and ask it to answer the question given the context provided by the search results. We can see an example in Figure 8-25.

This generation step is called *grounded generation* because the retrieved relevant information we provide the LLM establishes a certain context that grounds the LLM in the domain we're interested in. Figure 8-26 shows how grounded generation fits after search if we continue our embeddings search example from earlier.

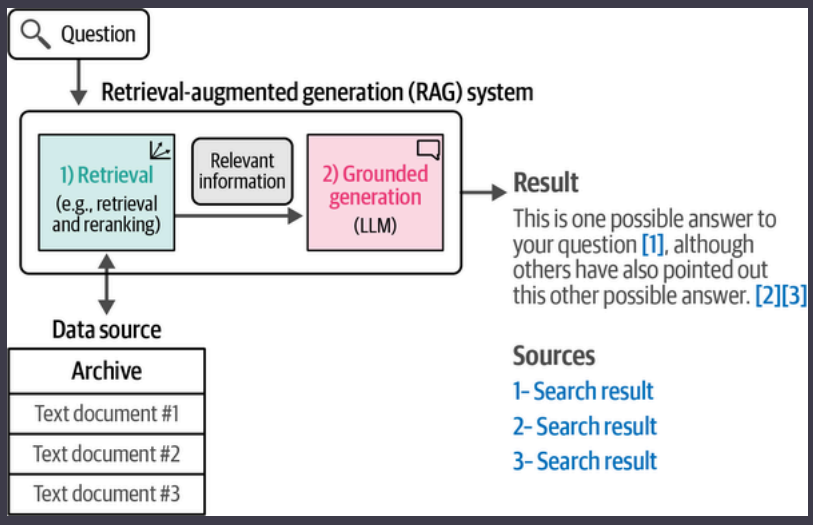

Figure 8-25. Generative search formulates answers and summaries at the end of a search pipeline while citing its sources (returned by the previous steps in the search system).

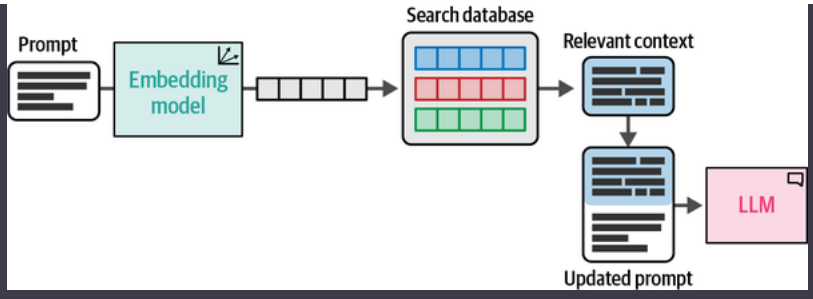

Figure 8-26. Find the most relevant information to an input prompt by comparing the similarities between embeddings. The most relevant information is added to the prompt before giving it to the LLM.

Example: Grounded Generation with an LLM API

Let's look at how to add a grounded generation step after the search results to create our first RAG system. For this example, we'll use Cohere's managed LLM, which builds on the search systems we've seen earlier in the chapter. We'll use embedding search to retrieve the top documents, then we'll pass those to the `co.chat` endpoint along with the questions to provide a grounded answer:

```
query = "income generated"

# 1- Retrieval
# We'll use embedding search. But ideally we'd do hybrid
results = search(query)

# 2- Grounded Generation
docs_dict = [{'text': text} for text in results['texts']]
response = co.chat(
    message = query,
    documents=docs_dict
)

print(response.text)
```

Result:

```
The film generated a worldwide gross of over $677 million, or $773 millio
```

We are highlighting some of the text because the model indicated the source for these spans of text to be the first document we passed in:

```
citations=[ChatCitation(start=21, end=36, text='worldwide gross', documen

documents=[{'id': 'doc_0', 'text': 'The film had a worldwide gross over $
```

Example: RAG with Local Models

Let us now replicate this basic functionality with local models. We will lose the ability to do span citations and the smaller local model isn't going to work as well as the larger managed model, but it's useful to demonstrate the flow. We'll start by downloading a quantized model.

Loading the generation model

We start by downloading our model:

```
!wget https://huggingface.co/microsoft/Phi-3-mini-4k-instruct-gguf/resolv
```

Using `llama.cpp`, `llama-cpp-python`, and LangChain, we load the text generation model:

```
from langchain import LlamaCpp

# Make sure the model path is correct for your system!
llm = LlamaCpp(
    model_path="Phi-3-mini-4k-instruct-fp16.gguf",
    n_gpu_layers=-1,
    max_tokens=500,
    n_ctx=2048,
    seed=42,
    verbose=False
)
```

Loading the embedding model

Let's now load an embedding language model. In this example, we will choose the BAAI/bge-small-en-v1.5 model. At the time of writing, it is high on the MTEB leaderboard for embedding models and relatively small:

```
from langchain.embeddings.huggingface import HuggingFaceEmbeddings

# Embedding model for converting text to numerical representations
embedding_model = HuggingFaceEmbeddings(
    model_name='thenlper/gte-small'
)
```

We can now use the embedding model to set up our vector database:

```
from langchain.vectorstores import FAISS

# Create a local vector database
db = FAISS.from_texts(texts, embedding_model)
```

The RAG prompt

A prompt template plays a vital part in the RAG pipeline. It is the central place where we communicate the relevant documents to the LLM. To do so, we will create an additional input variable named `context` that can provide the LLM with the retrieved documents:

```
from langchain import PromptTemplate

# Create a prompt template
template = """<|user|>
Relevant information:
{context}

Provide a concise answer the following question using the relevant information prov
{question}<|end|>
<|assistant|>"""
prompt = PromptTemplate(
    template=template,
    input_variables=["context", "question"]
)


from langchain.chains import RetrievalQA

# RAG pipeline
rag = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type='stuff',
    retriever=db.as_retriever(),
    chain_type_kwargs={
        "prompt": prompt
    },
    verbose=True
)
```

Now we're ready to call the model and ask it a question:

```
rag.invoke('Income generated')
```

Result:

```
The Income generated by the film in 2014 was over $677 million worldwide.
```

As always, we can adjust the prompt to control the model's generation (e.g., answer length and tone).

Advanced RAG Techniques

There are several additional techniques to improve the performance of RAG systems. Some of them are laid out here.

Query rewriting

If the RAG system is a chatbot, the preceding simple RAG implementation would likely struggle with the search step if a question is too verbose, or to refer to context in previous messages in the conversation. This is why it's a good idea to use an LLM to rewrite the query into one that aids the retrieval step in getting the right information. An example of this is a message such as:

> *User Question: "We have an essay due tomorrow. We have to write about some animal. I love penguins. I could write about them. But I could also write about dolphins. Are they animals? Maybe. Let's do dolphins. Where do they live for example?"*

This should actually be rewritten into a query like:

> *Query: "Where do dolphins live"*

This rewriting behavior can be done through a prompt (or through an API call). Cohere's API, for example, has a dedicated query-rewriting mode for `co.chat`.

Multi-query RAG

The next improvement we can introduce is to extend the query rewriting to be able to search multiple queries if more than one is needed to answer a specific question. Take for example:

> *User Question: "Compare the financial results of Nvidia in 2020 vs. 2023"*

We may find one document that contains the results for both years, but more likely, we're better off making two search queries:

> *Query 1: "Nvidia 2020 financial results"*

> *Query 2: "Nvidia 2023 financial results"*

We then present the top results of both queries to the model for grounded generation. An additional small improvement here is to also give the query rewriter the option to determine if no search is required and if it can directly generate a confident answer without searching.

Multi-hop RAG

A more advanced question may require a series of sequential queries. Take for example a question like:

> *User Question: "Who are the largest car manufacturers in 2023? Do they each make EVs or not?"*

To answer this, the system must first search for:

Step 1, Query 1: "largest car manufacturers 2023"

Then after it gets this information (the result being Toyota, Volkswagen, and Hyundai), it should ask follow-up questions:

Step 2, Query 1: "Toyota Motor Corporation electric vehicles"

Step 2, Query 2: "Volkswagen AG electric vehicles"

Step 2, Query 3: "Hyundai Motor Company electric vehicles"

Query routing

An additional enhancement is to give the model the ability to search multiple data sources. We can, for example, specify for the model that if it gets a question about HR, it should search the company's HR information system (e.g., Notion) but if the question is about customer data, that it should search the customer relationship management (CRM) (e.g., Salesforce).

Agentic RAG

You may be able to now see that the list of previous enhancements slowly delegates more and more responsibility to the LLM to solve more and more complex problems. This relies on the LLM's capability to gauge the required information needs as well as its ability to utilize multiple data sources. This new nature of the LLM starts to become closer and closer to an agent that acts on the world. The data sources can also now be abstracted into tools. We saw, for example, that we can search Notion, but by the same token, we should be able to post to Notion as well.

Not all LLMs will have the RAG capabilities mentioned here. At the time of writing, likely only the largest managed models may be able to attempt this behavior. Thankfully, Cohere's Command R+ excels at these tasks and is available as an open-weights model as well.

RAG Evaluation

There are still ongoing developments in how to evaluate RAG models. A good paper to read on this topic is "Evaluating verifiability in generative search engines" (2023), which runs human evaluations on different generative search systems.[2]

It evaluates results along four axes:

Fluency

: Whether the generated text is fluent and cohesive.

Perceived utility

: Whether the generated answer is helpful and informative.

Citation recall

: The proportion of generated statements about the external world that are fully supported by their citations.

Citation precision

The proportion of generated citations that support their associated statements.

While human evaluation is always preferred, there are approaches that attempt to automate these evaluations by having a capable LLM act as a judge (called *LLM-as-a-judge*) and score the different generations along the different axes. Ragas is a software library that does exactly this. It also scores some additional useful metrics like:

Faithfulness

Whether the answer is consistent with the provided context

Answer relevance

How relevant the answer is to the question

The Ragas documentation site provides more details about the formulas to actually calculate these metrics.

Summary

In this chapter, we looked at different ways of using language models to improve existing search systems and even be the core of new, more powerful search systems. These include:

- Dense retrieval, which relies on the similarity of text embeddings. These are systems that embed a search query and retrieve the documents with the nearest embeddings to the query's embedding.
- Rerankers, systems (like monoBERT) that look at a query and candidate results and score the relevance of each document to that query. These relevance scores are then used to order the shortlisted results according to their relevance to the query, often producing an improved results ranking.
- RAG, where search systems have a generative LLM at the end of the pipeline to formulate an answer based on retrieved documents while citing sources.

We also looked at one of the possible methods of evaluating search systems. Mean average precision allows us to score search systems to be able to compare across a test suite of queries and their known relevance to the test queries. Evaluating RAG systems requires multiple axes, however, like faithfulness, fluency, and others that can be evaluated by humans or by LLM-as-a-judge.

In the next chapter, we will explore how language models can be made multimodal and reason not just about text but images as well.

[1] Patrick Lewis et al. "Retrieval-augmented generation for knowledge-intensive NLP tasks." *Advances in Neural Information Processing Systems* 33 (2020): 9459–9474.

[2] Nelson F. Liu, Tianyi Zhang, and Percy Liang. "Evaluating verifiability in generative search engines." *arXiv preprint arXiv:2304.09848* (2023).